Copyrighted Materials
Copyright © 2014 Oxford University Press Retrieved from www.knovel.com

CHAPTER 3

Arrays

LEARNING OBJECTIVE

In this chapter, we will discuss arrays. An array is a user-defined data type that stores related information together. All the information stored in an array belongs to the same data type. So, in this chapter, we will learn how arrays are defined, declared, initialized, and accessed. We will also discuss the different operations that can be performed on array elements and the different types of arrays such as two-dimensional arrays, multi-dimensional arrays, and sparse matrices.

3.1 INTRODUCTION

We will explain the concept of arrays using an analogy. Consider a situation in which we have 20 students in a class and we have been asked to write a program that reads and prints the marks of all the 20 students. In this program, we will need 20 integer variables with different names, as shown in Fig. 3.1.

Now to read the values of these 20 variables, we must have 20 read statements. Similarly, to print the value of these variables, we need 20 write statements. If it is just a matter of 20 variables, then it might be acceptable for the user to follow this approach. But would it be possible to follow this approach if we have to read and print the marks of students,

- in the entire course (say 100 students)
- in the entire college (say 500 students)
- in the entire university (say 10,000 students)

The answer is no, definitely not! To process a large amount of data, we need a data structure known as *array*.

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an `index` (also known as the *subscript*). The subscript is an ordinal number which is used to identify an element of the array.

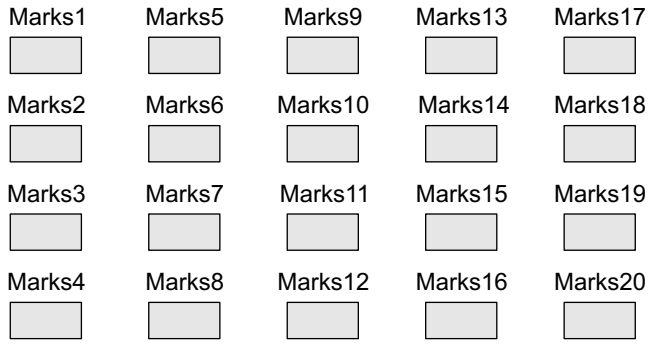

Figure 3.1 Twenty variables for 20 students

3.2 DECLARATION OF ARRAYS

We have already seen that every variable must be declared before it is used. The same concept holds true for array variables. An array must be declared before being used. Declaring an array means specifying the following:

- *Data type*—the kind of values it can store, for example, `int`, `char`, `float`, `double`.
- *Name*—to identify the array.
- *Size*—the maximum number of values that the array can hold.

Arrays are declared using the following syntax:

```
type name[size];
```

The type can be either `int`, `float`, `double`, `char`, or any other valid data type. The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array. For example, if we write,

```
int marks[10];
```

then the statement declares `marks` to be an array containing 10 elements. In C, the array index starts from zero. The first element will be stored in `marks[0]`, second element in `marks[1]`, and so on. Therefore, the last element, that is the 10th element, will be stored in `marks[9]`. Note that `0, 1, 2, 3` written within square brackets are the subscripts. In the memory, the array will be stored as shown in Fig. 3.2.

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Figure 3.2 Memory representation of an array of 10 elements

Figure 3.3 shows how different types of arrays are declared.

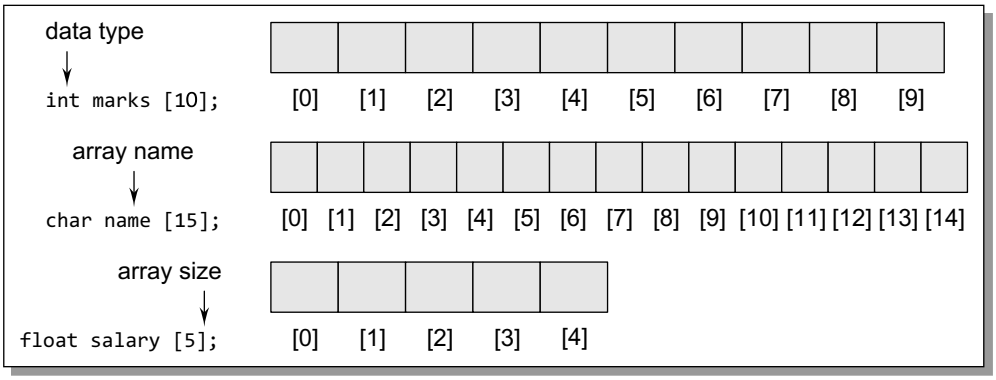

Figure 3.3 Declaring arrays of different data types and sizes

3.3 ACCESSING THE ELEMENTS OF AN ARRAY

Storing related data items in a single array enables the programmers to develop concise and efficient programs. But there is no single function that can operate on all the elements of an array. To access all the elements, we must use a loop. That is, we can access all the elements of an array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value. As shown in Fig. 3.2, the first element of the array `marks[10]` can be accessed by writing `marks[0]`. Now to process all the elements of the array, we use a loop as shown in Fig. 3.4.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
      marks[i] = -1;
```

Figure 3.4 Code to initialize each element of the array to –1

Figure 3.5 shows the result of the code shown in Fig. 3.4. The code accesses every individual element of the array and sets its value to –1. In the `for` loop, first the value of `marks[0]` is set to –1, then the value of the index `(i)` is incremented and the next value, that is, `marks[1]` is set to –1. The procedure continues until all the 10 elements of the array are set to –1.

– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

Note There is no single statement that can read, access, or print all the elements of an array. To do this, we have to use a loop to execute the same statement with different index values.

3.3.1 Calculating the Address of Array Elements

You must be wondering how C gets to know where an individual element of an array is located in the memory. The answer is that the array name is a symbolic reference to the address of the first byte of the array. When we use the array name, we are actually referring to the first byte of the array.

The subscript or the index represents the offset from the beginning of the array to the element being referenced. That is, with just the array name and the index, C can calculate the address of any element in the array.

Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient. The address of

other data elements can simply be calculated using the base address. The formula to perform this calculation is,

```
Address of data element, A[k] = BA(A) + w(k - lower_bound)
```

Here, `A` is the array, `k` is the index of the element of which we have to calculate the address, `BA` is the base address of the array `A`, and `w` is the size of one element in memory, for example, size of `int` is 2.

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
marks[4] = 1000 + 2(4 - 0)
         = 1000 + 2(4) = 1008
```

3.3.2 Calculating the Length of an Array

The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is

```
Length = upper_bound - lower_bound + 1
```

where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.

Example 3.2 Let `Age[5]` be an array of integers such that

```
Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
```

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

```
Length = upper_bound - lower_bound + 1
```

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 - 0 + 1 = 5`

3.4 STORING VALUES IN ARRAYS

When we declare an array, we are just allocating space for its elements; no values are stored in the array. There are three ways to store values in an array. First, to initialize the array elements during declaration; second, to input values for individual elements from the keyboard; third, to assign values to individual elements. This is shown in Fig. 3.6.

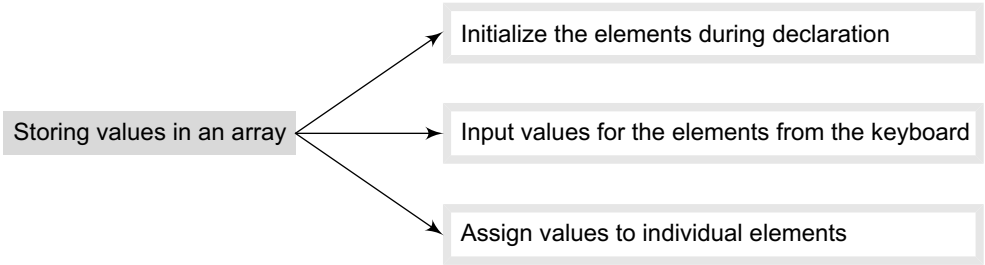

Figure 3.6 Storing values in an array

Initializing Arrays during Declaration

The elements of an array can be initialized at the time of declaration, just as any other variable. When an array is initialized, we need to provide a value for every element in the array. Arrays are initialized by writing,

```
type array_name[size]={list of values};
```

marks[0]	90
marks[1]	82
marks[2]	78
marks[3]	95
marks[4]	88

Figure 3.7 Initialization of array `marks[5]`

Note that the values are written within curly brackets and every value is separated by a comma. It is a compiler error to specify more values than there are elements in the array. When we write,

```
int marks[5]={90, 82, 78, 95, 88};
```

An array with the name `marks` is declared that has enough space to store five elements. The first element, that is, `marks[0]` is assigned value 90. Similarly, the second element of the array, that is `marks[1]`, is assigned 82, and so on. This is shown in Fig. 3.7.

While initializing the array at the time of declaration, the programmer may omit the size of the array. For example,

```
int marks[]= {98, 97, 90};
```

The above statement is absolutely legal. Here, the compiler will allocate enough space for all the initialized elements. Note that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros. Figure 3.8 shows the initialization of arrays.

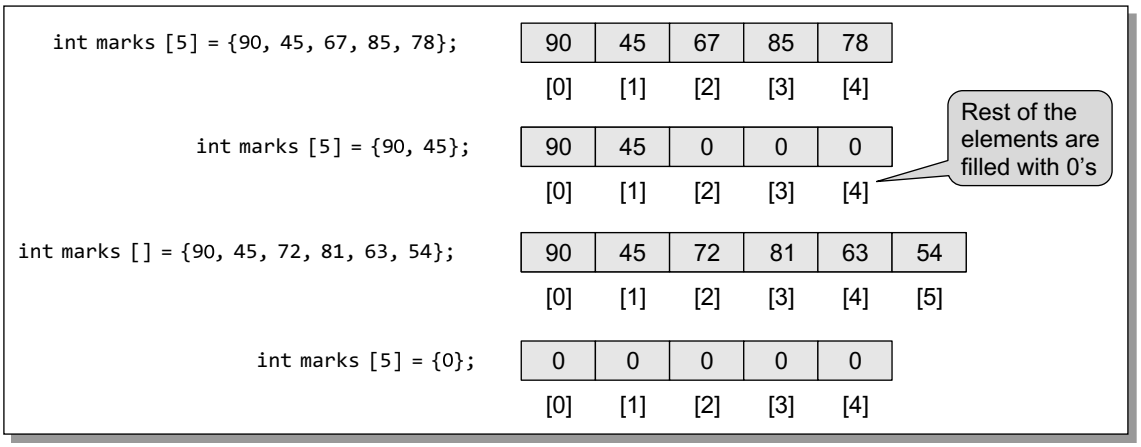

Figure 3.8 Initialization of array elements

```
int i, marks[10];
for(i=0;i<10;i++)
       scanf("%d", &marks[i]);
```

Figure 3.9 Code for inputting each element of the array

Inputting Values from the Keyboard

An array can be initialized by inputting values from the keyboard. In this method, a `while/do-while` or a `for` loop is executed to input the value for each element of the array. For example, look at the code shown in Fig. 3.9.

In the code, we start at the index `i` at 0 and input the value for the first element of the array. Since the array has 10 elements, we must input values for elements whose index varies from 0 to 9.

Assigning Values to Individual Elements

The third way is to assign values to individual elements of the array by using the assignment operator. Any value that evaluates to the data type as that of the array can be assigned to the individual array element. A simple assignment statement can be written as

```
marks[3] = 100;
```

Here, 100 is assigned to the fourth element of the array which is specified as `marks[3]`.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
       arr2[i] = arr1[i];
```

Figure 3.10 Code to copy an array at the individual element level

Note that we cannot assign one array to another array, even if the two arrays have the same type and size. To copy an array, you must copy the value of every element of the first array into the elements of the second array. Figure 3.10 illustrates the code to copy an array.

In Fig. 3.10, the loop accesses each element of the first array and simultaneously assigns its value to the corresponding element of the second array. The index value `i` is incremented to access the next element in succession. Therefore, when this code is executed, `arr2[0] = arr1[0]`, `arr2[1] = arr1[1]`, `arr2[2] = arr1[2]`, and so on.

```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
       arr[i] = i*2;
```

Figure 3.11 Code for filling an array with even numbers

We can also use a loop to assign a pattern of values to the array elements. For example, if we want to fill an array with even integers (starting from 0), then we will write the code as shown in Fig. 3.11.

In the code, we assign to each element a value equal to twice of its index, where the index starts from 0. So after executing this code, we will have `arr[0]=0`, `arr[1]=2`, `arr[2]=4`, and so on.

3.5 OPERATIONS ON ARRAYS

There are a number of operations that can be preformed on arrays. These operations include:

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

We will discuss all these operations in detail in this section, except searching and sorting, which will be discussed in Chapter 14.

3.5.1 Traversing an Array

Traversing an array means accessing each and every element of the array for a specific purpose.

Traversing the data elements of an array `A` can include printing every element, counting the total number of elements, or performing any process on these elements. Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward. The algorithm for array traversal is given in Fig. 3.12.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

In Step 1, we initialize the index to the lower bound of the array. In Step 2, a `while` loop is executed. Step 3 processes the individual array element as specified by the array name and index value. Step 4 increments the index value so that the next array element could be processed. The `while` loop in Step 2 is executed until all the elements in the array are processed, i.e., until `I` is less than or equal to the upper bound of the array.

Programming Examples

1. Write a program to read and display *n* numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d",&arr[i]);
        }
        printf("\n The array elements are ");
        for(i=0;i<n;i++)
                printf("\t %d", arr[i]);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are     1     2     3     4     5
```

2. Write a program to find the mean of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
```

```
{
        int i, n, arr[20], sum =0;
        float mean = 0.0;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d",&arr[i]);
        }
        for(i=0;i<n;i++)
                sum += arr[i];
        mean = (float)sum/n;
        printf("\n The sum of the array elements = %d", sum);
        printf("\n The mean of the array elements = %.2f", mean);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The sum of the array elements = 15
The mean of the array elements = 3.00
```

3. Write a program to print the position of the smallest number of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20], small, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i=0;i<n;i++)
                scanf("%d",&arr[i]);
        small = arr[0]
        pos =0;
        for(i=1;i<n;i++)
        {
                if(arr[i]<small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        printf("\n The smallest element is : %d", small);
        printf("\n The position of the smallest element in the array is : %d", pos);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements : 7 6 5 14 3
```

```
The smallest element is : 3
The position of the smallest element in the array is : 4
```

4. Write a program to find the second largest of *n* numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20], large, second_large;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&arr[i]);
        large = arr[0];
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                        large = arr[i];
        }
        second_large = arr[1];
        for(i=0;i<n;i++)
        {
                if(arr[i] != large)
                {
                        if(arr[i]>second_large)
                                second_large = arr[i];
                }
        }
        printf("\n The numbers you entered are : ");
        for(i=0;i<n;i++)
                printf("\t %d", arr[i]);
        printf("\n The largest of these numbers is : %d",large);
        printf("\n The second largest of these numbers is : %d",second_large);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements 1 2 3 4 5
The numbers you entered are :     1     2     3     4     5
The largest of these numbers is : 5
The second largest of these numbers is : 4
```

5. Write a program to enter *n* number of digits. Form a number using these digits.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
        int number=0, digit[10], numofdigits,i;
        clrscr();
        printf("\n Enter the number of digits : ");
        scanf("%d", &numofdigits);
        for(i=0;i<numofdigits;i++)
        {
                printf("\n Enter the digit at position %d", i+1);
```

```
                scanf("%d", &digit[i]);
        }
        i=0;
        while(i<numofdigits)
        {
                number = number + digit[i] * pow(10,i);
                i++;
        }
        printf("\n The number is : %d", number);
        return 0;
}
```

Output

```
Enter the number of digits : 4
Enter the digit at position 1: 2
Enter the digit at position 2 : 3
Enter the digit at position 3 : 0
Enter the digit at position 4 : 9
The number is : 9032
```

6. Write a program to find whether the array of integers contains a duplicate number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int array[10], i, n, j, flag =0;
        clrscr();
        printf("\n Enter the size of the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n array[%d] = ", i);
                scanf("%d", &array[i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(array[i] == array[j] && i!=j)
                        {
                        flag =1;
                        printf("\n Duplicate numbers found at locations %d and %d", i, j);
                        }
                }
        }
        if(flag==0)
                printf("\n No Duplicates Found");
        return 0;
}
```

Output

```
Enter the size of the array : 5
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 2
array[4] = 5
Duplicate numbers found at locations 1 and 3
```

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Figure 3.13 Algorithm to append a new element to an existing array

3.5.2 Inserting an Element in an Array

If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. We just have to add 1 to the `upper_bound` and assign the value. Here, we assume that the memory space allocated for the array is still available. For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

Figure 3.13 shows an algorithm to insert a new element to the end of an array. In Step 1, we increment the value of the `upper_bound`. In Step 2, the new value is stored at the position pointed by the `upper_bound`. For example, let us assume an array has been declared as

```
int marks[60];
```

The array is declared to store the marks of all the students in a class. Now, suppose there are 54 students and a new student comes and is asked to take the same test. The marks of this new student would be stored in `marks[55]`. Assuming that the student secured 68 marks, we will assign the value as

```
marks[55] = 68;
```

However, if we have to insert an element in the middle of the array, then this is not a trivial task. On an average, we might have to move as much as half of the elements from their positions in order to accommodate space for the new element.

For example, consider an array whose elements are arranged in ascending order. Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array. To do this, we must first find the location where the new element will be inserted and then move all the elements (that have a value greater than that of the new element) one position to the right so that space can be created to store the new value.

Example 3.3 `Data[]` is an array that is declared as `int Data[20];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) Calculate the length of the array.
(b) Find the `upper_bound` and `lower_bound`.
(c) Show the memory representation of the array.
(d) If a new data element with the value 75 has to be inserted, find its position.
(e) Insert a new data element 75 and show the memory representation after the insertion.

Solution

(a) Length of the array = number of elements
Therefore, `length of the array = 10`

(b) By default, `lower_bound = 0` and `upper_bound = 9`

(c)

12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one position towards the right to accommodate the new value

(e)

12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

Algorithm to Insert an Element in the Middle of an Array

The algorithm `INSERT` will be declared as `INSERT (A, N, POS, VAL)`. The arguments are

(a) `A`, the array in which the element has to be inserted
(b) `N`, the number of elements in the array
(c) `POS`, the position at which the element has to be inserted
(d) `VAL`, the value that has to be inserted

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:         SET A[I + 1] = A[I]
Step 4:         SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

In the algorithm given in Fig. 3.14, in Step 1, we first initialize I with the total number of elements in the array. In Step 2, a `while` loop is executed which will move all the elements having an index greater than `POS` one position towards right to create space for the new element. In Step 5, we increment the total number of elements in the array by 1 and finally in Step 6, the new value is inserted at the desired position.

Now, let us visualize this algorithm by taking an example.

Initial `Data[]` is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling `INSERT(Data, 6, 3, 100)` will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

PROGRAMMING EXAMPLES

7. Write a program to insert a number at a given location in an array.

```
#include <stdio.h>
```

```
#include <conio.h>
int main()
{
        int i, n, num, pos, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number to be inserted : ");
        scanf("%d", &num);
        printf("\n Enter the position at which the number has to be added :");
        scanf("%d", &pos);
        for(i=n-1;i>=pos;i--)
                arr[i+1] = arr[i];
        arr[pos] = num;
        n = n+1;
        printf("\n The array after insertion of %d is : ", num);
        for(i=0;i<n;i++)
            printf("\n arr[%d] = %d", i, arr[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
Enter the number to be inserted : 0
Enter the position at which the number has to be added : 3
The array after insertion of 0 is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 0
arr[4] = 4
arr[5] = 5
```

8. Write a program to insert a number in an array that is already sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, j, num, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number to be inserted : ");
```

```
        scanf("%d", &num);
        for(i=0;i<n;i++)
        {
                if(arr[i] > num)
                {
                        for(j = n-1; j>=i; j--)
                                arr[j+1] = arr[j];
                        arr[i] = num;
                        break;
                }
        }
        n = n+1;
        printf("\n The array after insertion of %d is : ", num);
        for(i=0;i<n;i++)
                printf("\n arr[%d] = %d", i, arr[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 4
arr[3] = 5
arr[4] = 6
Enter the number to be inserted : 3
The array after insertion of 3 is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
```

3.5.3 Deleting an Element from an Array

Deleting an element from an array means removing a data element from an already existing array. If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the `upper_bound`. Figure 3.15 shows an algorithm to delete an element from the end of an array.

For example, if we have an array that is declared as

```
int marks[60];
```

The array is declared to store the marks of all the students in the class. Now, suppose there are 54 students and the student with roll number 54 leaves the course. The score of this student was stored in `marks[54]`. We just have to decrement the `upper_bound`. Subtracting 1 from the `upper_bound` will indicate that there are 53 valid data in the array.

However, if we have to delete an element from the middle of an array, then it is not a trivial task. On an average, we might have to move as much as half of the elements from their positions in order to occupy the space of the deleted element.

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Figure 3.15 Algorithm to delete the last element of an array

For example, consider an array whose elements are arranged in ascending order. Now, suppose an element has to be deleted, probably from somewhere in the

middle of the array. To do this, we must first find the location from where the element has to be deleted and then move all the elements (having a value greater than that of the element) one position towards left so that the space vacated by the deleted element can be occupied by rest of the elements.

Example 3.4 `Data[]` is an array that is declared as `int Data[10];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) If a data element with value 56 has to be deleted, find its position.

(b) Delete the data element 56 and show the memory representation after the deletion.

Solution

(a) Since the elements of the array are stored in ascending order, we will compare the value that has to be deleted with the value of every element in the array. As soon as `VAL = Data[I]`, where `I` is the index or subscript of the array, we will get the position from which the element has to be deleted. For example, if we see this array, here `VAL = 56`. `Data[0] = 12` which is not equal to 56. We will continue to compare and finally get the value of `POS = 4`.

(b)

12	23	34	45	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:         SET A[I] = A[I + 1]
Step 4:         SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Figure 3.16 Algorithm to delete an element from the middle of an array

Algorithm to Delete an Element from the Middle of an Array

The algorithm `DELETE` will be declared as `DELETE(A, N, POS)`. The arguments are:

(a) `A`, the array from which the element has to be deleted

(b) `N`, the number of elements in the array

(c) `POS`, the position from which the element has to be deleted

Figure 3.16 shows the algorithm in which we first initialize `I` with the position from which the element has to be deleted. In Step 2, a `while` loop is executed which will move all the elements having an index greater than `POS` one space towards left to occupy the space vacated by the deleted element. When we say that we are deleting an element, actually we are overwriting the element with the value of its successive element. In Step 5, we decrement the total number of elements in the array by 1.

Now, let us visualize this algorithm by taking an example given in Fig. 3.17. Calling `DELETE(Data, 6, 2)` will lead to the following processing in the array.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

Figure 3.17 Deleting elements from an array

PROGRAMMING EXAMPLE

9. Write a program to delete a number from a given location in an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, pos, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &arr[i]);
        }
        printf("\nEnter the position from which the number has to be deleted : ");
        scanf("%d", &pos);
        for(i=pos; i<n-1;i++)
                arr[i] = arr[i+1];
        n--;
        printf("\n The array after deletion is : ");
        for(i=0;i<n;i++)
                printf("\n arr[%d] = %d", i, arr[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
Enter the position from which the number has to be deleted : 3
The array after deletion is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 5
```

10. Write a program to delete a number from an array that is already sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, j, num, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number to be deleted : ");
        scanf("%d", &num);
```

```
        for(i=0;i<n;i++)
        {
                if(arr[i] == num)
                {
                        for(j=i; j<n-1;j++)
                                arr[j] = arr[j+1];
                }
        }
        n = n-1;
        printf("\n The array after deletion is : ");
        for(i=0;i<n;i++)
                printf("\n arr[%d] = %d", i, arr[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
Enter the number to be deleted : 3
The array after deletion is :
arr[0] = 1
arr[1] = 2
arr[2] = 4
arr[3] = 5
```

3.5.4 Merging Two Arrays

Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.

If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another. But merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted. Let us first discuss the merge operation on unsorted arrays. This operation is shown in Fig 3.18.

Array 1-	90	56	89	77	69							
Array 2-	45	88	76	99	12	58	81					
Array 3-	90	56	89	77	69	45	88	76	99	12	58	81

Figure 3.18 Merging of two unsorted arrays

PROGRAMMING EXAMPLE

11. Write a program to merge two unsorted arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
```

```
{
        int arr1[10], arr2[10], arr3[20];
        int i, n1, n2, m, index=0;
        clrscr();
        printf("\n Enter the number of elements in array1 : ");
        scanf("%d", &n1);
        printf("\n\n Enter the elements of the first array");
        for(i=0;i<n1;i++)
        {
                printf("\n arr1[%d] = ", i);
                scanf("%d", &arr1[i]);
        }
        printf("\n Enter the number of elements in array2 : ");
        scanf("%d", &n2);
        printf("\n\n Enter the elements of the second array");
        for(i=0;i<n2;i++)
        {
                printf("\n arr2[%d] = ", i);
                scanf("%d", &arr2[i]);
        }
        m = n1+n2;
        for(i=0;i<n1;i++)
        {
            arr3[index] = arr1[i];
            index++;
        }
        for(i=0;i<n2;i++)
        {
                arr3[index] = arr2[i];
                index++;
        }
        printf("\n\n The merged array is");
        for(i=0;i<m;i++)
                printf("\n arr[%d] = %d", i, arr3[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in array1 : 3
Enter the elements of the first array
arr1[0] = 1
arr1[1] = 2
arr1[2] = 3
Enter the number of elements in array2 : 3
Enter the elements of the second array
arr2[0] = 4
arr2[1] = 5
arr2[2] = 6
The merged array is
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
```

If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult. The task of merging can be explained using Fig. 3.19.

Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

Figure 3.19 Merging of two sorted arrays

Figure 3.19 shows how the merged array is formed using two sorted arrays. Here, we first compare the 1st element of `array1` with the 1st element of `array2`, and then put the smaller element in the merged array. Since 20 > 15, we put 15 as the first element in the merged array. We then compare the 2nd element of the second array with the 1st element of the first array. Since 20 < 22, now 20 is stored as the second element of the merged array. Next, the 2nd element of the first array is compared with the 2nd element of the second array. Since 30 > 22, we store 22 as the third element of the merged array. Now, we will compare the 2nd element of the first array with the 3rd element of the second array. Because 30 < 31, we store 30 as the 4th element of the merged array. This procedure will be repeated until elements of both the arrays are placed in the right location in the merged array.

Programming Example

12. Write a program to merge two sorted arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int arr1[10], arr2[10], arr3[20];
        int i, n1, n2, m, index=0;
        int index_first = 0, index_second = 0;
        clrscr();
        printf("\n Enter the number of elements in array1 : ");
        scanf("%d", &n1);
        printf("\n\n Enter the elements of the first array");
        for(i=0;i<n1;i++)
        {
                printf("\n arr1[%d] = ", i);
                scanf("%d", &arr1[i]);
        }
        printf("\n Enter the number of elements in array2 : ");
        scanf("%d", &n2);
        printf("\n\n Enter the elements of the second array");
        for(i=0;i<n2;i++)
        {
                printf("\n arr2[%d] = ", i);
                scanf("%d", &arr2[i]);
        }
        m = n1+n2;
        while(index_first < n1 && index_second < n2)
        {
```

```
                if(arr1[index_first]<arr2[index_second])
                {
                        arr3[index] = arr1[index_first];
                        index_first++;
                }
                else
                {
                        arr3[index] = arr2[index_second];
                        index_second++;
                }
                index++;
        }
        // if elements of the first array are over and the second array has some elements
        if(index_first == n1)
        {
                while(index_second<n2)
                {
                        arr3[index] = arr2[index_second];
                        index_second++;
                        index++;
                }
        }
        // if elements of the second array are over and the first array has some elements
        else if(index_second == n2)
        {
                while(index_first<n1)
                {
                        arr3[index] = arr1[index_first];
                        index_first++;
                        index++;
                }
        }
        printf("\n\n The merged array is");
        for(i=0;i<m;i++)
                printf("\n arr[%d] = %d", i, arr3[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of elements in array1 : 3
Enter the elements of the first array
arr1[0] = 1
arr1[1] = 3
arr1[2] = 5
Enter the number of elements in array2 : 3
Enter the elements of the second array
arr2[0] = 2
arr2[1] = 4
arr2[2] = 6
The merged array is
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
```

3.6 PASSING ARRAYS TO FUNCTIONS

Like variables of other data types, we can also pass an array to a function. In some situations, you may want to pass individual elements of the array; while in other situations, you may want to pass the entire array. In this section, we will discuss both the cases. Look at Fig. 3.20 which will help you understand the concept.

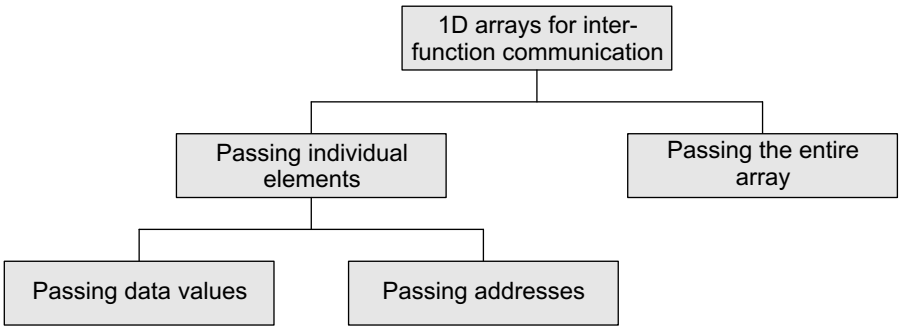

Figure 3.20 One dimensional arrays for inter-function communication

3.6.1 Passing Individual Elements

The individual elements of an array can be passed to a function by passing either their data values or addresses.

Passing Data Values

Individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match with the type of the function parameter. Look at Fig. 3.21(a) which shows the code to pass an individual array element by passing the data value.

Calling function	Called function

`main()`
`{`
`  int arr[5] ={1, 2, 3, 4, 5};`
`  func(arr[3]);`
`}`
	`void func(int num)`
	`{`
	`  printf("%d", num);`
	`}`

Figure 3.21(a) Passing values of individual array elements to a function

In the above example, only one element of the array is passed to the called function. This is done by using the index expression. Here, `arr[3]` evaluates to a single integer value. The called function hardly bothers whether a normal integer variable is passed to it or an array value is passed.

Passing Addresses

Like ordinary variables, we can pass the address of an individual array element by preceding the indexed array element with the address operator. Therefore, to pass the address of the fourth element of the array to the called function, we will write `&arr[3]`.

However, in the called function, the value of the array element must be accessed using the indirection `(*)` operator. Look at the code shown in Fig. 3.21(b).

Calling function	Called function

`main()`
`{`
`int arr[5] ={1, 2, 3, 4, 5};`
`func(&arr[3]);`
`}`
	`void func(int *num)`
	`{`
	`printf("%d", *num);`
	`}`

Figure 3.21(b) Passing addresses of individual array elements to a function

3.6.2 Passing the Entire Array

We have discussed that in C the array name refers to the first byte of the array in the memory. The address of the remaining elements in the array can be calculated using the array name and the index value of the element. Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array. Figure 3.22 illustrates the code which passes the entire array to the called function.

Calling function	Called function

`main()`
`{`
`int arr[5] ={1, 2, 3, 4, 5};`
`func(arr);`
`}`
	`void func(int arr[5])`
	`{`
	`int i;`
	`for(i=0;i<5;i++)`
	`printf("%d", arr[i]);`
	`}`

Figure 3.22 Passing entire array to a function

A function that accepts an array can declare the formal parameter in either of the two following ways.

`func(int arr[]);` or `func(int *arr);`

When we pass the name of an array to a function, the address of the zeroth element of the array is copied to the local pointer variable in the function. When a formal parameter is declared in a function header as an array, it is interpreted as a pointer to a variable and not as an array. With this pointer variable you can access all the elements of the array by using the expression: `array_name + index`. You can also pass the size of the array as another parameter to the function. So for a function that accepts an array as parameter, the declaration should be as follows.

`func(int arr[], int n); or func(int *arr, int n);`

It is not necessary to pass the whole array to a function. We can also pass a part of the array known as a sub-array. A pointer to a sub-array is also an array pointer. For example, if we want to send the array starting from the third element then we can pass the address of the third element and the size of the sub-array, i.e., if there are 10 elements in the array, and we want to pass the array starting from the third element, then only eight elements would be part of the sub-array. So the function call can be written as

`func(&arr[2], 8);`

Note that in case we want the called function to make no changes to the array, the array must be received as a constant array by the called function. This prevents any type of unintentional modifications of the array elements. To declare an array as a constant array, simply add the keyword `const` before the data type of the array.

Look at the following programs which illustrate the use of pointers to pass an array to a function.

Programming Examples

13. Write a program to read an array of *n* numbers and then find the smallest number.

```
#include <stdio.h>
#include <conio.h>
void read_array(int arr[], int n);
int find_small(int arr[], int n);
int main()
{
        int num[10], n, smallest;
        clrscr();
        printf("\n Enter the size of the array : ");
        scanf("%d", &n);
        read_array(num, n);
        smallest = find_small(num, n);
        printf("\n The smallest number in the array is = %d", smallest);
        getch();
        return 0;
}
void read_array(int arr[10], int n)
{
        int i;
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &arr[i]);
        }
}
int find_small(int arr[10], int n)
{
        int i = 0, small = arr[0];
        for(i=1;i<n;i++)
        {
                if(arr[i] < small)
                        small = arr[i];
        }
        return small;
}
```

Output

```
Enter the size of the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The smallest number in the array is = 1
```

14. Write a program to interchange the largest and the smallest number in an array.

```
#include <stdio.h>
#include <conio.h>
void read_array(int my_array[], int);
void display_array(int my_array[], int);
void interchange(int arr[], int);
int find_biggest_pos(int my_array[10], int n);
int find_smallest_pos(int my_array[10], int n);
int main()
{
```

```
        int arr[10], n;
        clrscr();
        printf("\n Enter the size of the array : ");
        scanf("%d", &n);
        read_array(arr, n);
        interchange(arr, n);
        printf("\n The new array is: ");
        display_array(arr,n);
        getch();
        return 0;
}
void read_array(int my_array[10], int n)
{
        int i;
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d", &my_array[i]);
        }
}
void display_array(int my_array[10], int n)
{
        int i;
        for(i=0;i<n;i++)
                printf("\n arr[%d] = %d", i, my_array[i]);
}
void interchange(int my_array[10], int n)
{
        int temp, big_pos, small_pos;
        big_pos = find_biggest_pos(my_array, n);
        small_pos = find_smallest_pos(my_array,n);
        temp = my_array[big_pos];
        my_array[big_pos] = my_array[small_pos];
        my_array[small_pos] = temp;
}
int find_biggest_pos(int my_array[10], int n)
{
        int i, large = my_array[0], pos=0;
        for(i=1;i<n;i++)
        {
                if (my_array[i] > large)
                {
                        large = my_array[i];
                        pos=i;
                }
        }
        return pos;
}
int find_smallest_pos (int my_array[10], int n)
{
        int i, small = my_array[0], pos=0;
        for(i=1;i<n;i++)
        {
                if (my_array[i] < small)
                {
                        small = my_array[i];
                        pos=i;
                }
        }
```

```
            return pos;
    }
```

Output

```
Enter the size of the array : 5
arr[0] = 5
arr[1] = 1
arr[2] = 6
arr[3] = 3
arr[4] = 2
The new array is :
arr[0] = 5
arr[1] = 6
arr[2] = 1
arr[3] = 3
arr[4] = 2
```

3.7 POINTERS AND ARRAYS

The concept of array is very much bound to the concept of pointer. Consider Fig. 3.23. For example, if we have an array declared as,

```
int arr[] = {1, 2, 3, 4, 5};
```

then in memory it would be stored as shown in Fig. 3.23.

1	2	3	4	5
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1000	1002	1004	1006	1008

Figure 3.23 Memory representation of `arr[]`

Array notation is a form of pointer notation. The name of the array is the starting address of the array in memory. It is also known as the base address. In other words, base address is the address of the first element in the array or the address of `arr[0]`. Now let us use a pointer variable as given in the statement below.

```
int *ptr;
ptr = &arr[0];
```

Programming Tip

The name of an array is actually a pointer that points to the first element of the array.

Here, `ptr` is made to point to the first element of the array. Execute the code given below and observe the output which will make the concept clear to you.

```
main()
{
    int arr[]={1,2,3,4,5};
    printf("\n Address of array = %p %p %p", arr, &arr[0], &arr);
}
```

Similarly, writing `ptr = &arr[2]` makes `ptr` to point to the third element of the array that has index `2`. Figure 3.24 shows `ptr` pointing to the third element of the array.

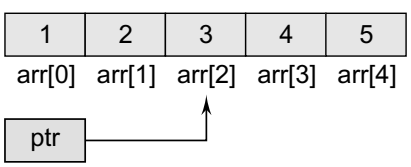

Figure 3.24 Pointer pointing to the third element of the array

If pointer variable `ptr` holds the address of the first element in the array, then the address of successive elements can be calculated by writing `ptr++`.

```
int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d",
*ptr);
```

Programming Tip

An error is generated if an attempt is made to change the address of the array.

The `printf()` function will print the value `2` because after being incremented `ptr` points to the next location. One point to note here is that if `x` is an integer variable, then `x++;` adds `1` to the value of `x`. But `ptr`

is a pointer variable, so when we write `ptr+i`, then adding `i` gives a pointer that points `i` elements further along an array than the original pointer.

Since `++ptr` and `ptr++` are both equivalent to `ptr+1`, incrementing a pointer using the unary `++` operator, increments the address it stores by the amount given by `sizeof(type)` where `type` is the data type of the variable it points to (i.e., `2` for an integer). For example, consider Fig. 3.25.

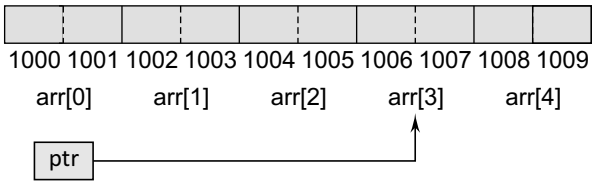

Figure 3.25 Pointer (ptr) pointing to the fourth element of the array

Programming Tip

When an array is passed to a function, we are actually passing a pointer to the function. Therefore, in the function declaration you must declare a pointer to receive the array name.

If `ptr` originally points to `arr[2]`, then `ptr++` will make it to point to the next element, i.e., `arr[3]`. This is shown in Fig. 3.25.

Had this been a character array, every byte in the memory would have been used to store an individual character. `ptr++` would then add only 1 byte to the address of `ptr`.

When using pointers, an expression like `arr[i]` is equivalent to writing `*(arr+i)`.

Many beginners get confused by thinking of array name as a pointer. For example, while we can write

```
ptr = arr;   // ptr = &arr[0]
```

we cannot write

```
arr = ptr;
```

This is because while `ptr` is a variable, `arr` is a constant. The location at which the first element of `arr` will be stored cannot be changed once `arr[]` has been declared. Therefore, an array name is often known to be a constant pointer.

To summarize, the name of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the element that it points to. Therefore, arrays and pointers use the same concept.

Note arr[i], i[arr], *(arr+i), *(i+arr) gives the same value.

Look at the following code which modifies the contents of an array using a pointer to an array.

```
int main()
{
      int arr[]={1,2,3,4,5};
      int *ptr, i;
      ptr=&arr[2];
      *ptr = -1;
      *(ptr+1) = 0;
      *(ptr-1) = 1;
      printf("\n Array is: ");
      for(i=0;i<5;i++)
            printf(" %d", *(arr+i));
      return 0;
}
```

Output

```
Array is: 1 1 -1 0 5
```

In C we can add or subtract an integer from a pointer to get a new pointer, pointing somewhere other than the original position. C also permits addition and subtraction of two pointer variables. For example, look at the code given below.

```
int main()
{
      int arr[]={1,2,3,4,5,6,7,8,9};
      int *ptr1, *ptr2;
      ptr1 = arr;
      ptr2 = arr+2;
      printf("%d", ptr2-ptr1);
      return 0;
}
```

Output

```
2
```

In the code, `ptr1` and `ptr2` are pointers pointing to the elements of the same array. We may subtract two pointers as long as they point to the same array. Here, the output is `2` because there are two elements between `ptr1` and `ptr2` in the array `arr`. Both the pointers must point to the same array or one past the end of the array, otherwise this behaviour cannot be defined.

Moreover, C also allows pointer variables to be compared with each other. Obviously, if two pointers are equal, then they point to the same location in the array. However, if one pointer is less than the other, it means that the pointer points to some element nearer to the beginning of the array. Like with other variables, relational operators (`>, <, >=,` etc.) can also be applied to pointer variables.

PROGRAMMING EXAMPLE

15. Write a program to display an array of given numbers.

```
#include <stdio.h>
int main()
{
        int arr[]={1,2,3,4,5,6,7,8,9};
        int *ptr1, *ptr2;
        ptr1 = arr;
        ptr2 = &arr[8];
        while(ptr1<=ptr2)
        {
                printf("%d", *ptr1);
                ptr1++;
        }
        return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9
```

3.8 ARRAYS OF POINTERS

An array of pointers can be declared as

```
int *ptr[10];
```

The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];
int p = 1, q = 2, r = 3, s = 4, t = 5;
ptr[0] = &p;
ptr[1] = &q;
ptr[2] = &r;
ptr[3] = &s;
ptr[4] = &t;
```

Can you tell what will be the output of the following statement?

```
printf("\n %d", *ptr[3]);
```

The output will be `4` because `ptr[3]` stores the address of integer variable `s` and `*ptr[3]` will therefore print the value of s that is `4`. Now look at another code in which we store the address of three individual arrays in the array of pointers:

```
int main()
{
      int arr1[]={1,2,3,4,5};
      int arr2[]={0,2,4,6,8};
      int arr3[]={1,3,5,7,9};
      int *parr[3] = {arr1, arr2, arr3};
      int i;
      for(i = 0;i<3;i++)
            printf(«%d», *parr[i]);
      return 0;
}
```

Output

```
1 0 1
```

Surprised with this output? Try to understand the concept. In the `for` loop, `parr[0]` stores the base address of `arr1` (or, `&arr1[0]`). So writing `*parr[0]` will print the value stored at `&arr1[0]`. Same is the case with `*parr[1]` and `*parr[2]`.

3.9 TWO-DIMENSIONAL ARRAYS

Till now, we have only discussed one-dimensional arrays. One-dimensional arrays are organized linearly in only one direction. But at times, we need to store data in the form of grids or tables. Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures. A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column. The C compiler treats a two-dimensional array as an array of one-dimensional arrays. Figure 3.26 shows a two-dimensional array which can be viewed as an array of arrays.

3.9.1 Declaring Two-dimensional Arrays

Any array must be declared before being used. The declaration statement tells the compiler the name of the array, the data type of each element in the array, and the size of each dimension. A two-dimensional array is declared as:

```
data_type array_name[row_size][column_size];
```

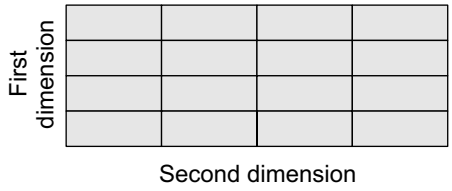

Figure 3.26 Two-dimensional array

Therefore, a two-dimensional `m×n` array is an array that contains `m × n` data elements and each element is accessed using two subscripts, `i` and `j`, where `i<=m` and `j<=n`.

For example, if we want to store the marks obtained by three students in five different subjects, we can declare a two-dimensional array as:

```
int marks[3][5];
```

In the above statement, a two-dimensional array called `marks` has been declared that has `m(3)` rows and `n(5)` columns. The first element of the array is denoted by `marks[0][0]`, the second element as `marks[0][1]`, and so on. Here, `marks[0][0]` stores the marks obtained by the first student in the first subject, `marks[1][0]` stores the marks obtained by the second student in the first subject.

The pictorial form of a two-dimensional array is shown in Fig. 3.27.

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	`marks[0][0]`	`marks[0][1]`	`marks[0][2]`	`marks[0][3]`	`marks[0][4]`
Row 1	`marks[1][0]`	`marks[1][1]`	`marks[1][2]`	`marks[1][3]`	`marks[1][4]`
Row 2	`marks[2][0]`	`marks[2][1]`	`marks[2][2]`	`marks[2][3]`	`marks[2][4]`

Figure 3.27 Two-dimensional array

Hence, we see that a 2D array is treated as a collection of 1D arrays. Each row of a 2D array corresponds to a 1D array consisting of `n` elements, where `n` is the number of columns. To understand this, we can also see the representation of a two-dimensional array as shown in Fig. 3.28.

`marks[0]` –	`marks[0]`	`marks[1]`	`marks[2]`	`marks[3]`	`marks[4]`
`marks[1]` –	`marks[0]`	`marks[1]`	`marks[2]`	`marks[3]`	`marks[4]`
`marks[2]` –	`marks[0]`	`marks[1]`	`marks[2]`	`marks[3]`	`marks[4]`

Figure 3.28 Representation of two-dimensional array `marks[3][5]`

Although we have shown a rectangular picture of a two-dimensional array, in the memory, these elements actually will be stored sequentially. There are two ways of storing a two-dimensional array in the memory. The first way is the *row major order* and the second is the *column major order*. Let us see how the elements of a 2D array are stored in a row major order. Here, the elements of the first row are stored before the elements of the second and third rows. That is, the elements of the array are stored row by row where `n` elements of the first row will occupy the first `n` locations. This is illustrated in Fig. 3.29.

(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)

Figure 3.29 Elements of a 3×4 2D array in row major order

However, when we store the elements in a column major order, the elements of the first column are stored before the elements of the second and third column. That is, the elements of the array are stored column by column where `m` elements of the first column will occupy the first `m` locations. This is illustrated in Fig. 3.30.

(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)

Figure 3.30 Elements of a 4×3 2D array in column major order

In one-dimensional arrays, we have seen that the computer does not keep track of the address of every element in the array. It stores only the address of the first element and calculates the address of other elements from the base address (address of the first element). Same is the case with a two-dimensional array. Here also, the computer stores the base address, and the address of the other elements is calculated using the following formula.

If the array elements are stored in column major order,

```
Address(A[I][J]) = Base_Address + w{M ( J – 1) + (I – 1)}
```

And if the array elements are stored in row major order,

```
Address(A[I][J]) = Base_Address + w{N ( I – 1) + (J – 1)}
```

where `w` is the number of bytes required to store one element, `N` is the number of columns, `M` is the number of rows, and `I` and `J` are the subscripts of the array element.

Example 3.5 Consider a 20×5 two-dimensional array `marks` which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J]) = Base_Address + w{N (I – 1) + (J – 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 – 1) + (4 – 1)}
                      = 1000 + 2 {5(17) + 3}
                      = 1000 + 2 (88)
                      = 1000 + 176 = 1176
```

3.9.2 Initializing Two-dimensional Arrays

Like in the case of other variables, declaring a two-dimensional array only reserves space for the array in the memory. No values are stored in it. A two-dimensional array is initialized in the same way as a one-dimensional array is initialized. For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};
```

Note that the initialization of a two-dimensional array is done row by row. The above statement can also be written as:

```
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

The above two-dimensional array has two rows and three columns. First, the elements in the first row are initialized and then the elements of the second row are initialized.

Therefore,

```
marks[0][0] =   90      marks[0][1] =   87      marks[0][2] =   78
marks[1][0] =   68      marks[1][1] =   62      marks[1][2] =   71
```

In the above example, each row is defined as a one-dimensional array of three elements that are enclosed in braces. Note that the commas are used to separate the elements in the row as well as to separate the elements of two rows.

In case of one-dimensional arrays, we have discussed that if the array is completely initialized, we may omit the size of the array. The same concept can be applied to a two-dimensional array, except that only the size of the first dimension can be omitted. Therefore, the declaration statement given below is valid.

```
int marks[][3]={{90,87,78},{68, 62, 71}};
```

In order to initialize the entire two-dimensional array to zeros, simply specify the first value as zero. That is,

```
int marks[2][3] = {0};
```

The individual elements of a two-dimensional array can be initialized using the assignment operator as shown here.

```
marks[1][2] = 79;
```

or

```
marks[1][2] = marks[1][1] + 10;
```

3.9.3 Accessing the Elements of Two-dimensional Arrays

The elements of a 2D array are stored in contiguous memory locations. In case of one-dimensional arrays, we used a single `for` loop to vary the index `i` in every pass, so that all the elements could be scanned. Since the two-dimensional array contains two subscripts, we will use two `for` loops to scan the elements. The first `for` loop will scan each row in the 2D array and the second `for` loop will scan individual columns for every row in the array. Look at the programs which use two `for` loops to access the elements of a 2D array.

PROGRAMMING EXAMPLES

16. Write a program to print the elements of a 2D array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int arr[2][2] = {12, 34, 56,32};
        int i, j;
        for(i=0;i<2;i++)
        {
              printf("\n");
              for(j=0;j<2;j++)
                      printf("%d\t", arr[i][j]);
        }
        return 0;
}
```

Output

```
12      34
56      32
```

17. Write a program to generate Pascal's triangle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int arr[7][7]={0};
        int row=2, col, i, j;
        arr[0][0] = arr[1][0] = arr[1][1] = 1;
        while(row <= 7)
        {
                arr[row][0] = 1;
                for(col = 1; col <= row; col++)
                arr[row][col] = arr[row-1][col-1] + arr[row-1][col];
                row++;
        }
        for(i=0; i<7; i++)
        {
                printf("\n");
                for(j=0; j<=i; j++)
```

```
                        printf("\t %d", arr[i][j]);
        }
        getch();
        return 0;
}
```

Output

```
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
1       6       15      20      15      6       1
```

18. In a small company there are five salesmen. Each salesman is supposed to sell three products. Write a program using a 2D array to print (i) the total sales by each salesman and (ii) total sales of each item.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int sales[5][3], i, j, total_sales=0;
        //INPUT DATA
        printf("\n ENTER THE DATA");
        printf("\n *****************");
        for(i=0; i<5; i++)
        {
                printf("\n Enter the sales of 3 items sold by salesman %d: ", i+1);
                for(j=0; j<3; j++)
                        scanf("%d", &sales[i][j]);
        }
        // PRINT TOTAL SALES BY EACH SALESMAN
        for(i=0; i<5; i++)
        {
                total_sales = 0;
                for(j=0; j<3; j++)
                        total_sales += sales[i][j];
                printf("\n Total Sales By Salesman %d = %d", i+1, total_sales);
        }
        // TOTAL SALES OF EACH ITEM
        for(i=0; i<3; i++)// for each item
        {
                total_sales=0;
                for(j=0; j<5; j++)// for each salesman
                        total_sales += sales[j][i];
                printf("\n Total sales of item %d = %d", i+1, total_sales);
        }
        getch();
        return 0;
}
```

Output

```
ENTER THE DATA
*****************
Enter the sales of 3 items sold by salesman 1: 23 23 45
Enter the sales of 3 items sold by salesman 2: 34 45 63
Enter the sales of 3 items sold by salesman 3: 36 33 43
Enter the sales of 3 items sold by salesman 4: 33 52 35
```

```
Enter the sales of 3 items sold by salesman 5: 32 45 64
Total Sales By Salesman 1 = 91
Total Sales By Salesman 2 = 142
Total Sales By Salesman 3 = 112
Total Sales By Salesman 4 = 120
Total Sales By Salesman 5 = 141
Total sales of item 1 = 158
Total sales of item 2 = 198
Total sales of item 3 = 250
```

19. Write a program to read a 2D array `marks` which stores the marks of five students in three subjects. Write a program to display the highest marks in each subject.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int marks[5][3], i, j, max_marks;
        for(i=0; i<5; i++)
        {
                printf("\n Enter the marks obtained by student %d",i+1);
                for(j=0; j<3; j++)
                {
                        printf("\n marks[%d][%d] = ", i, j);
                        scanf("%d", &marks[i][j]);
                }
        }
        for(j=0; j<3; j++)
        {
                max_marks = -999;
                for(i=0; i<5; i++)
                {
                        if(marks[i][j]>max_marks)
                        max_marks = marks[i][j];
                }
                printf("\n The highest marks obtained in the subject %d = %d", j+1, max_marks);
        }
        getch();
        return 0;
}
```

Output

```
Enter the marks obtained by student 1
marks[0][0] = 89
marks[0][1] = 76
marks[0][2] = 100
Enter the marks obtained by student 2
marks[1][0] = 99
marks[1][1] = 90
marks[1][2] = 89
Enter the marks obtained by student 3
marks[2][0] = 67
marks[2][1] = 76
marks[2][2] = 56
Enter the marks obtained by student 4
marks[3][0] = 88
marks[3][1] = 77
marks[3][2] = 66
Enter the marks obtained by student 5
```

```
marks[4][0] = 67
marks[4][1] = 78
marks[4][2] = 89
The highest marks obtained in the subject 1 = 99
The highest marks obtained in the subject 2 = 90
The highest marks obtained in the subject 3 = 100
```

3.10 OPERATIONS ON TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays can be used to implement the mathematical concept of matrices. In mathematics, a matrix is a grid of numbers, arranged in rows and columns. Thus, using two-dimensional arrays, we can perform the following operations on an $m \times n$ matrix:

Transpose Transpose of an $m \times n$ matrix A is given as a $n \times m$ matrix B, where $B_{i,j} = A_{j,i}$.

Sum Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$$C_{i,j} = A_{i,j} + B_{i,j}$$

Difference Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing:

$$C_{i,j} = A_{i,j} - B_{i,j}$$

Product Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, $m \times n$ matrix A can be multiplied with a $p \times q$ matrix B if $n=p$. The dimension of the product matrix is $m \times q$. The elements of two matrices can be multiplied by writing:

$$C_{i,j} = \Sigma A_{i,k} B_{k,j} \text{ for } k=1 \text{ to } n$$

PROGRAMMING EXAMPLES

20. Write a program to read and display a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j, mat[3][3];
        clrscr();
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                    scanf("%d",&mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
```

```
                                printf("\t %d",mat[i][j]);
            }
            return 0;
    }
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
```

21. Write a program to transpose a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j, mat[3][3], transposed_mat[3][3];
        clrscr();
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d", &mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                        printf("\t %d", mat[i][j]);
        }
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                        transposed_mat[i][j] = mat[j][i];
        }
        printf("\n The elements of the transposed matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                        printf("\t %d",transposed_ mat[i][j]);
        }
        return 0;
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
The elements of the transposed matrix are
1 4 7
2 5 8
3 6 9
```

22. Write a program to input two `m × n` matrices and then calculate the sum of their corresponding elements and store it in a third `m × n` matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j;
        int rows1, cols1, rows2, cols2, rows_sum, cols_sum;
        int mat1[5][5], mat2[5][5], sum[5][5];
        clrscr();
        printf("\n Enter the number of rows in the first matrix : ");
        scanf("%d",&rows1);
        printf("\n Enter the number of columns in the first matrix : ");
        scanf("%d",&cols1);
        printf("\n Enter the number of rows in the second matrix : ");
        scanf("%d",&rows2);
        printf("\n Enter the number of columns in the second matrix : ");
        scanf("%d",&cols2);
        if(rows1 != rows2 || cols1 != cols2)
        {
             printf("\n Number of rows and columns of both matrices must be equal");
             getch();
             exit();
        }
        rows_sum = rows1;
        cols_sum = cols1;
        printf("\n Enter the elements of the first matrix ");
        for(i=0;i<rows1;i++)
        {
                for(j=0;j<cols1;j++)
                {
                        scanf("%d",&mat1[i][j]);
                }
        }
        printf("\n Enter the elements of the second matrix ");
        for(i=0;i<rows2;i++)
        {
                for(j=0;j<cols2;j++)
                {
                        scanf("%d",&mat2[i][j]);
                }
        }
        for(i=0;i<rows_sum;i++)
        {
                for(j=0;j<cols_sum;j++)
                        sum[i][j] = mat1[i][j] + mat2[i][j];
        }
        printf("\n The elements of the resultant matrix are ");
        for(i=0;i<rows_sum;i++)
        {
                printf("\n");
                for(j=0;j<cols_sum;j++)
                        printf("\t %d", sum[i][j]);
        }
        return 0;
}
```

Output

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the resultant matrix are
6 8
10 12
```

23. Write a program to multiply two `m` × `n` matrices.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j, k;
        int rows1, cols1, rows2, cols2, res_rows, res_cols;
        int mat1[5][5], mat2[5][5], res[5][5];
        clrscr();
        printf("\n Enter the number of rows in the first matrix : ");
        scanf("%d",&rows1);
        printf("\n Enter the number of columns in the first matrix : ");
        scanf("%d",&cols1);
        printf("\n Enter the number of rows in the second matrix : ");
        scanf("%d",&rows2);
        printf("\n Enter the number of columns in the second matrix : ");
        scanf("%d",&cols2);
        if(cols1 != rows2)
        {
                printf("\n The number of columns in the first matrix must be equal
                to the number of rows in the second matrix");
                getch();
                exit();
        }
        res_rows = rows1;
        res_cols = cols2;
        printf("\n Enter the elements of the first matrix ");
        for(i=0;i<rows1;i++)
        {
                for(j=0;j<cols1;j++)
                {
                        scanf("%d",&mat1[i][j]);
                }
        }
        printf("\n Enter the elements of the second matrix ");
        for(i=0;i<rows2;i++)
        {
                for(j=0;j<cols2;j++)
                {
                        scanf("%d",&mat2[i][j]);
                }
        }
        for(i=0;i<res_rows;i++)
        {
                for(j=0;j<res_cols;j++)
```

```
                {
                        res[i][j]=0;
                        for(k=0; k<res_cols;k++)
                        res[i][j] += mat1[i][k] * mat2[k][j];
                }
        }
        printf("\n The elements of the product matrix are ");
        for(i=0;i<res_rows;i++)
        {
                printf("\n");
                for(j=0;j<res_cols;j++)
                        printf("\t %d",res[i][j]);
        }
        return 0;
}
```

Output

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the product matrix are
19 22
43 50
```

3.11 PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

There are three ways of passing a two-dimensional array to a function. First, we can pass individual elements of the array. This is exactly the same as passing an element of a one-dimensional array. Second, we can pass a single row of the two-dimensional array. This is equivalent to passing the entire one-dimensional array to a function that has already been discussed in a previous section. Third, we can pass the entire two-dimensional array to the function. Figure 3.31 shows the three ways of using two-dimensional arrays for inter-functon communication.

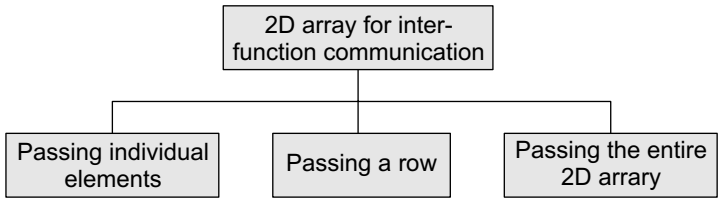

Figure 3.31 2D arrays for inter-function communication

Passing a Row

A row of a two-dimensional array can be passed by indexing the array name with the row number. Look at Fig. 3.32 which illustrates how a single row of a two-dimensional array can be passed to the called function.

Calling function	Called function

`main()`
`{`
`  int arr[2][3] = ({1, 2, 3}, {4, 5, 6});`
`  func(arr[1]);`
`}`
	`void func(int arr[])`
	`{`
	`  int i;`
	`  for(i=0;i<3;i++)`
	`  printf("%d", arr[i] * 10);`
	`}`

Figure 3.32 Passing a row of a 2D array to a function

Passing the Entire 2D Array

To pass a two-dimensional array to a function, we use the array name as the actual parameter (the way we did in case of a 1D array). However, the parameter in the called function must indicate that the array has two dimensions. Look at the following program which passes entire 2D array to a function.

PROGRAMMING EXAMPLE

24. Write a program to fill a square matrix with value zero on the diagonals, 1 on the upper right triangle, and –1 on the lower left triangle.

```
#include <stdio.h>
#include <conio.h>
void read_matrix(int mat[5][5], int);
void display_matrix(int mat[5][5], int);
int main()
{
        int row;
        int mat1[5][5];
        clrscr();
        printf("\n Enter the number of rows and columns of the matrix:");
        scanf("%d", &row);
        read_matrix(mat1, row);
        display_matrix(mat1, row);
        getch();
        return 0;
}

void read_matrix(int mat[5][5], int r)
{
        int i, j;
        for(i=0; i<r; i++)
        {
              for(j=0; j<r; j++)
              {
                      if(i==j)
                          mat[i][j] = 0;
                      else if(i>j)
                          mat[i][j] = -1;
                      else
                          mat[i][j] = 1;
              }
        }
}
void display_matrix(int mat[5][5], int r)
{
        int i, j;
```

```
            for(i=0; i<r; i++)
            {
            printf("\n");
            for(j=0; j<r; j++)
                    printf("\t %d", mat[i][j]);
            }
    }
```

Output

```
Enter the number of rows and columns of the matrix: 2
 0               1
-1               0
```

3.12 POINTERS AND TWO-DIMENSIONAL ARRAYS

Consider a two-dimensional array declared as

```
int mat[5][5];
```

To declare a pointer to a two-dimensional array, you may write

```
int **ptr
```

Here `int **ptr` is an array of pointers (to one-dimensional arrays), while `int mat[5][5]` is a 2D array. They are not the same type and are not interchangeable.

Individual elements of the array `mat` can be accessed using either:

```
mat[i][j] or
*(*(mat + i) + j) or
*(mat[i]+j);
```

To understand more fully the concept of pointers, let us replace
`*(multi + row)` with `X` so the expression
`*(*(mat + i) + j)` becomes `*(X + col)`

Using pointer arithmetic, we know that the address pointed to by (i.e., value of) `X + col + 1` must be greater than the address `X + col` by an amount equal to `sizeof(int)`.

Since `mat` is a two-dimensional array, we know that in the expression `multi + row` as used above, `multi + row + 1` must increase in value by an amount equal to that needed to *point to* the next row, which in this case would be an amount equal to `COLS * sizeof(int)`.

Thus, in case of a two-dimensional array, in order to evaluate expression (for a row major 2D array), we must know a total of 4 values:

1. The address of the first element of the array, which is given by the name of the array, i.e., `mat` in our case.
2. The size of the type of the elements of the array, i.e., size of integers in our case.
3. The specific index value for the row.
4. The specific index value for the column.

Note that

```
int (*ptr)[10];
```

declares `ptr` to be a pointer to an array of 10 integers. This is different from

```
int *ptr[10];
```

which would make `ptr` the name of an array of 10 pointers to type `int`. You must be thinking how pointer arithmetic works if you have an array of pointers. For example:

```
int * arr[10] ;
int ** ptr = arr ;
```

In this case, `arr` has type `int **`. Since all pointers have the same size, the address of `ptr + i` can be calculated as:

```
addr(ptr + i) = addr(ptr) + [sizeof(int *) * i]
= addr(ptr) + [2 * i]
```

Since `arr` has type `int **`,

```
arr[0] = &arr[0][0],
arr[1] = &arr[1][0], and in general,
arr[i] = &arr[i][0].
```

According to pointer arithmetic, `arr + i = & arr[i]`, yet this skips an entire row of 5 elements, i.e., it skips complete 10 bytes (5 elements each of 2 bytes size). Therefore, if `arr` is address **1000**, then `arr + 2` is address **1010**. To summarize, `&arr[0][0]`, `arr[0]`, `arr`, and `&arr[0]` point to the base address.

```
&arr[0][0] + 1 points to arr[0][1]
arr[0] + 1 points to arr[0][1]
arr + 1 points to arr[1][0]
&arr[0] + 1 points to arr[1][0]
```

To conclude, a two-dimensional array is not the same as an array of pointers to 1D arrays. Actually a two-dimensional array is declared as:

```
int (*ptr)[10] ;
```

Here `ptr` is a pointer to an array of 10 elements. The parentheses are not optional. In the absence of these parentheses, `ptr` becomes an array of 10 pointers, not a pointer to an array of 10 ints.

Look at the code given below which illustrates the use of a pointer to a two-dimensional array.

```
#include <stdio.h>
int main()
{
      int arr[2][2]={{1,2}, {3,4}};
      int i, (*parr)[2];
      parr = arr;
      for(i = 0; i < 2; i++)
      {
            for(j = 0; j < 2 ;j++)
                  printf(" %d", (*(parr+i))[j]);
      }
      return 0;
}
```

Output

```
1  2  3  4
```

The golden rule to access an element of a two-dimensional array can be given as

```
arr[i][j] = (*(arr+i))[j] = *((*arr+i))+j) = *(arr[i]+j)
```

Therefore,

```
arr[0][0] = *(arr)[0] = *((*arr)+0) = *(arr[0]+0)
arr[1][2] = (*(arr+1))[2] = *((*(arr+1))+2) = *(arr[1]+2)
```

If we declare an array of pointers using, `data_type *array_name[SIZE];` Here `SIZE` represents the number of rows and the space for columns that can be dynamically allocated.	If we declare a pointer to an array using, `data_type (*array_name)[SIZE];` Here `SIZE` represents the number of columns and the space for rows that may be dynamically allocated (refer Appendix A to see how memory is dynamically allocated).

PROGRAMMING EXAMPLE

25. Write a program to read and display a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
void display(int (*)[3]);
int main()
{
        int i, j, mat[3][3];
        clrscr();
        printf("\n Enter the elements of the matrix");
        for(i=0;i<3;i++)
         {
                    for(j = 0; j < 3; j++)
                    {
                          scanf("%d", &mat[i][j]);
                    }
        }
        display(mat);
        return 0;
}
void display(int (*mat)[3])
{
        int i, j;
        printf("\n The elements of the matrix are");
        for(i = 0; i < 3; i++)
        {
        printf("\n");
        for(j=0;j<3;j++)
                printf("\t %d",*(*(mat + i)+j));
        }
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
```

Note A double pointer cannot be used as a 2D array. Therefore, it is wrong to declare: 'int **mat' and then use 'mat' as a 2D array. These are two very different data types used to access different locations in memory. So running such a code may abort the program with a 'memory access violation' error.

A 2D array is not equivalent to a double pointer. A 'pointer to pointer of T' cannot serve as a '2D array of T'. The 2D array is equivalent to a pointer to row of T, and this is very different from pointer to pointer of T.

When a double pointer that points to the first element of an array is used with the subscript notation ptr[0][0], it is fully dereferenced two times and the resulting object will have an address equal to the value of the first element of the array

3.13 MULTI-DIMENSIONAL ARRAYS

A multi-dimensional array in simple terms is an array of arrays. As we have one index in a one-dimensional array, two indices in a two-dimensional array, in the same way, we have n indices in an n-dimensional array or multi-dimensional array. Conversely, an n-dimensional array is specified

using n indices. An n-dimensional $m_1 \times m_2 \times m_3 \times \cdots \times m_n$ array is a collection of $m_1 \times m_2 \times m_3 \times \cdots \times m_n$ elements. In a multi-dimensional array, a particular element is specified by using n subscripts as $A[I_1][I_2][I_3]...[I_n]$, where

$I_1 <= M_1,\ I_2 <= M_2,\ I_3 <= M_3,\ ...I_n <= M_n$

A multi-dimensional array can contain as many indices as needed and as the requirement of memory increases with the number of indices used. However, in practice, we hardly use more than three indices in any program. Figure 3.33 shows a three-dimensional array. The array has three pages, four rows, and two columns.

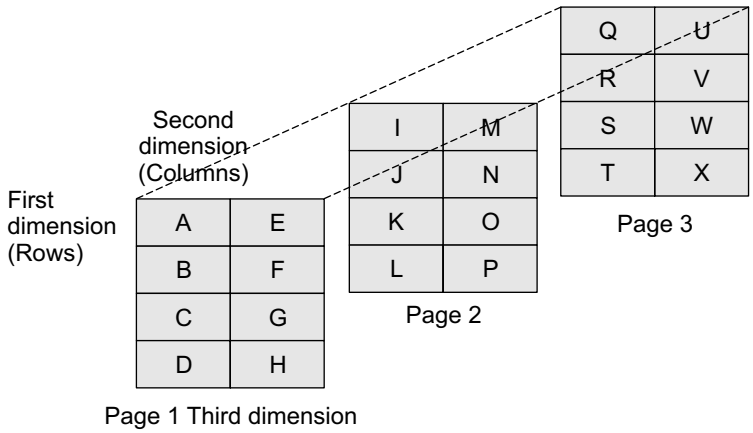

Figure 3.33 Three-dimensional array

Note A multi-dimensional array is declared and initialized the same way we declare and initialize one- and two-dimensional arrays.

Example 3.6 Consider a three-dimensional array defined as `int A[2][2][3]`. Calculate the number of elements in the array. Also, show the memory representation of the array in the row major order and the column major order.

Solution

A three-dimensional array consists of pages. Each page, in turn, contains m rows and n columns.

(0,0,0)	(0,0,1)	(0,0,2)	(0,1,0)	(0,1,1)	(0,1,2)	(1,0,0)	(1,0,1)	(1,0,2)	(1,1,0)	(1,1,1)	(1,1,2)

(a) Row major order

(0,0,0)	(0,1,0)	(0,0,1)	(0,1,1)	(0,0,2)	(0,1,2)	(1,0,0)	(1,1,0)	(1,0,1)	(1,1,1)	(1,0,2)	(1,1,2)

(b) Column major order

The three-dimensional array will contain $2 \times 2 \times 3 = 12$ elements.

PROGRAMMING EXAMPLE

26. Write a program to read and display a 2×2×2 array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int array[2][2][2], i, j, k;
        clrscr();
        printf("\n Enter the elements of the matrix");
        for(i=0;i<2;i++)
        {
                for(j=0;j<2;j++)
                {
                        for(k=0;k<2;k++)
                        {
                                scanf("%d", &array[i][j][k]);
                        }
                }
        }
        printf("\n The matrix is : ");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                {
                        printf("\n");
                        for(k=0;k<2;k++)
                                printf("\t array[%d][%d][%d] = %d", i, j, k, array[i]
[j][k]);
                }
        }
        getch();
        return 0;
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8
The matrix is
arr[0][0][0] = 1 arr[0][0][1] = 2
arr[0][1][0] = 3 arr[0][1][1] = 4
arr[1][0][0] = 5 arr[1][0][1] = 6
arr[1][1][0] = 7 arr[1][1][1] = 8
```

3.14 POINTERS AND THREE-DIMENSIONAL ARRAYS

In this section, we will see how pointers can be used to access a three-dimensional array. We have seen that pointer to a one-dimensional array can be declared as,

```
int arr[]={1,2,3,4,5};
int *parr;
parr = arr;
```

Similarly, pointer to a two-dimensional array can be declared as,

```
int arr[2][2]={{1,2},{3,4}};
int (*parr)[2];
parr = arr;
```

A pointer to a three-dimensional array can be declared as,

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};
int (*parr)[2][2];
parr = arr;
```

We can access an element of a three-dimensional array by writing,

```
arr[i][j][k] = *(*(*(arr+i)+j)+k)
```

PROGRAMMING EXAMPLE

27. Write a program which illustrates the use of a pointer to a three-dimensional array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i,j,k;
        int arr[2][2][2];
        int (*parr)[2][2]= arr;
        clrscr();
        printf("\n Enter the elements of a 2 × 2 × 2 array: ");
        for(i = 0; i < 2; i++)
        {
                for(j = 0; j < 2; j++)
                {
                        for(k = 0; k < 2; k++)
                                scanf("%d", &arr[i][j][k]);
                }
        }
        printf("\n The elements of the 2 × 2 × 2 array are: ");
        for(i = 0; i < 2; i++)
        {
                for(j = 0; j < 2; j++)
                {
                        for(k = 0; k < 2; k++)
                                printf("%d", *(*(*(parr+i)+j)+k));
                }
        }
        getch();
        return 0;
}
```

Output

```
Enter the elements of a 2 × 2 × 2 array: 1 2 3 4 5 6 7 8
The elements of the 2 × 2 × 2 array are: 1 2 3 4 5 6 7 8
```

Note In the printf statement, you could also have used `*(*(*(arr+i)+j+)+k)` instead of `*(*(*(parr+i)+j)+k))`.

3.15 SPARSE MATRICES

Sparse matrix is a matrix that has large number of elements with a zero value. In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used. If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory. Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

$$\begin{bmatrix} 1 & & & & \\ 5 & 3 & & & \\ 2 & 7 & -1 & & \\ 3 & 1 & 4 & 2 & \\ -9 & 2 & -8 & 1 & 7 \end{bmatrix}$$

Figure 3.34 Lower-triangular matrix

There are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also called a (*lower*) *triagonal matrix* because if you see it pictorially, all the elements with a non-zero value appear below the diagonal. In a lower triangular matrix, $A_{i,j}=0$ where `i<j`. An `n×n` lower-triangular matrix `A` has one non-zero element in the first row, two non-zero elements in the second row and likewise `n` non-zero elements in the *n*th row. Look at Fig. 3.34 which shows a lower-triangular matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 3 & 6 & 7 & 8 \\ & & -1 & 9 & 1 \\ & & & 9 & 2 \\ & & & & 7 \end{bmatrix}$$

Figure 3.35 Upper-triangular matrix

To store a lower-triangular matrix efficiently in the memory, we can use a one-dimensional array which stores only non-zero elements. The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

(a) Row-wise mapping—Here the contents of array `A[]` will be {1, 5, 3, 2, 7, –1, 3, 1, 4, 2, –9, 2, –8, 1, 7}

(b) Column-wise mapping—Here the contents of array `A[]` will be {1, 5, 2, 3, –9, 3, 7, 1, 2, –1, 4, –8, 2, 1, 7}

In an *upper-triangular matrix*, $A_{i,j}=0$ where `i>j`. An `n×n` upper-triangular matrix `A` has `n` non-zero elements in the first row, `n-1` non-zero elements in the second row and likewise one non-zero element in the *n*th row. Look at Fig. 3.35 which shows an upper-triangular matrix.

$$\begin{bmatrix} 4 & 1 & & & & \\ 5 & 1 & 2 & & & \\ & 9 & 3 & 1 & & \\ & & 4 & 2 & 2 & \\ & & & 5 & 1 & 9 \\ & & & & 8 & 7 \end{bmatrix}$$

Figure 3.36 Tri-diagonal matrix

There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a *tri-diagonal matrix*. Hence in a tridiagonal matrix, $A_{i,j}=0$, where `|i - j| > 1`. In a tridiagonal matrix, if elements are present on

(a) the main diagonal, it contains non-zero elements for `i=j`. In all, there will be `n` elements.
(b) below the main diagonal, it contains non-zero elements for `i=j+1`. In all, there will be `n-1` elements.
(c) above the main diagonal, it contains non-zero elements for `i=j-1`. In all, there will be `n-1` elements.

Figure 3.36 shows a tri-diagonal matrix. To store a tri-diagonal matrix efficiently in the memory, we can use a one-dimensional array that stores only non-zero elements. The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

(a) Row-wise mapping—Here the contents of array `A[]` will be {4, 1, 5, 1, 2, 9, 3, 1, 4, 2, 2, 5, 1, 9, 8, 7}
(b) Column-wise mapping—Here the contents of array `A[]` will be {4, 5, 1, 1, 9, 2, 3, 4, 1, 2, 5, 2, 1, 8, 9, 7}
(c) Diagonal-wise mapping—Here the contents of array `A[]` will be {5, 9, 4, 5, 8, 4, 1, 3, 2, 1, 7, 1, 2, 1, 2, 9}

3.16 APPLICATIONS OF ARRAYS

Arrays are frequently used in C, as they have a number of useful applications. These applications are

- Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables. We will read about these data structures in the subsequent chapters.
- Arrays can be used for sorting elements in ascending or descending order.

Points to Remember

- An array is a collection of elements of the same data type.
- The elements of an array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- The index specifies an offset from the beginning of the array to the element being referenced.
- Declaring an array means specifying three parameters: data type, name, and its size.
- The length of an array is given by the number of elements stored in it.
- There is no single function that can operate on all the elements of an array. To access all the elements, we must use a loop.
- The name of an array is a symbolic reference to the address of the first byte of the array. Therefore, whenever we use the array name, we are actually referring to the first byte of that array.
- C considers a two-dimensional array as an array of one-dimensional arrays.
- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second subscript denotes the column of the array.
- Using two-dimensional arrays, we can perform the different operations on matrices: transpose, addition, subtraction, multiplication.
- A multi-dimensional array is an array of arrays. Like we have one index in a one-dimensional array, two indices in a two-dimensional array, in the same way we have *n* indices in an *n*-dimensional or multi-dimensional array. Conversely, an *n*-dimensional array is specified using *n* indices.
- Multi-dimensional arrays can be stored in either row major order or column major order.
- Sparse matrix is a matrix that has large number of elements with a zero value.
- There are two types of sparse matrices. In the first type, all the elements above the main diagonal have a zero value. This type of sparse matrix is called a lower-triangular matrix. In the second type, all the elements below the main diagonal have a zero value. This type of sparse matrix is called an upper-triangular matrix.
- There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of sparse matrix is called a tridiagonal matrix.

Exercises

Review Questions

1. What are arrays and why are they needed?
2. How is an array represented in the memory?
3. How is a two-dimensional array represented in the memory?
4. What is the use of multi-dimensional arrays?
5. Explain sparse matrix.
6. How are pointers used to access two-dimensional arrays?
7. Why does storing of sparse matrices need extra consideration? How are sparse matrices stored efficiently in the computer's memory?
8. For an array declared as `int arr[50]`, calculate the address of `arr[35]`, if `Base(arr)=1000` and `w=2`.
9. Consider a two-dimensional array `Marks[10][5]` having its base address as 2000 and the number of bytes per element of the array is 2. Now, compute the address of the element, `Marks[8][5]`, assuming that the elements are stored in row major order.
10. How are arrays related to pointers?
11. Briefly explain the concept of array of pointers.
12. How can one-dimensional arrays be used for inter-function communication?
13. Consider a two-dimensional array `arr[10][10]` which has base address = 1000 and the number of bytes per element of the array = 2. Now, compute the address of the element `arr[8][5]` assuming that the elements are stored in column major order.
14. Consider the array given below:

`Name[0]`	Adam
`Name[1]`	Charles
`Name[2]`	Dicken
`Name[3]`	Esha
`Name[4]`	Georgia
`Name[5]`	Hillary
`Name[6]`	Mishael

(a) How many elements would be moved if the name Andrew has to be added in it?
(i) 7 (ii) 4
(iii) 5 (iv) 6

(b) How many elements would be moved if the name Esha has to be deleted from it?
(i) 3 (ii) 4
(iii) 5 (iv) 6

15. What happens when an array is initialized with
(a) fewer initializers as compared to its size?
(b) more initializers as compared to its size?

Programming Exercises

1. Consider an array `MARKS[20][5]` which stores the marks obtained by 20 students in 5 subjects. Now write a program to
(a) find the average marks obtained in each subject.
(b) find the average marks obtained by every student.
(c) find the number of students who have scored below 50 in their average.
(d) display the scores obtained by every student.

2. Write a program that reads an array of 100 integers. Display all the pairs of elements whose sum is 50.

3. Write a program to interchange the second element with the second last element.

4. Write a program that calculates the sum of squares of the elements.

5. Write a program to compute the sum and mean of the elements of a two-dimensional array.

6. Write a program to read and display a square (using functions).

7. Write a program that computes the sum of the elements that are stored on the main diagonal of a matrix using pointers.

8. Write a program to add two 3 × 3 matrix using pointers.

9. Write a program that computes the product of the elements that are stored on the diagonal above the main diagonal.

10. Write a program to count the total number of non-zero elements in a two-dimensional array.

11. Write a program to input the elements of a two-dimensional array. Then from this array, make two arrays—one that stores all odd elements of the two-dimensional array and the other that stores all even elements of the array.

12. Write a program to read two floating point number arrays. Merge the two arrays and display the resultant array in reverse order.

13. Write a program using pointers to interchange the second biggest and the second smallest number in the array.

14. Write a menu driven program to read and display a `p×q×r` matrix. Also, find the sum, transpose, and product of the two `p×q×r` matrices.

15. Write a program that reads a matrix and displays the sum of its diagonal elements.

16. Write a program that reads a matrix and displays the sum of the elements above the main diagonal. (*Hint: Calculate the sum of elements* A_{ij} *where* `i<j`)

17. Write a program that reads a matrix and displays the sum of the elements below the main diagonal. (*Hint: Calculate the sum of elements* A_{ij} *where* `i>j`)

18. Write a program that reads a square matrix of size `n×n`. Write a function `int isUpperTriangular (int a[][], int n)` that returns 1 if the matrix is upper triangular.
(*Hint: Array* `A` *is upper triangular if* A_{ij} `= 0` *and* `i>j`)

19. Write a program that reads a square matrix of size `n×n`. Write a function `int isLowerTriangular (int a[][], int n)` that returns 1 if the matrix is lower triangular.
(*Hint: Array* `A` *is lower triangular if* A_{ij} `= 0` *and* `i<j`)

20. Write a program that reads a square matrix of size `n×n`. Write a function `int isSymmetric (int a[][], int n)` that returns 1 if the matrix is symmetric. (*Hint: Array* `A` *is symmetric if* $A_{ij} = A_{ji}$ *for all values of* `i` *and* `j`)

21. Write a program to calculate `XA + YB` where `A` and `B` are matrices and `X=2` and `Y=3`.

22. Write a program to illustrate the use of a pointer that points to a 2D array.

23. Write a program to enter a number and break it into `n` number of digits.

24. Write a program to delete all the duplicate entries from an array of `n` integers.

25. Write a program to read a floating point array. Update the array to insert a new number at the specified location.

Multiple-choice Questions

1. If an array is declared as `arr[] = {1,3,5,7,9};` then what is the value of `sizeof(arr[3])`?
   (a) 1 (b) 2
   (c) 3 (d) 8
2. If an array is declared as `arr[] = {1,3,5,7,9};` then what is the value of `arr[3]`?
   (a) 1 (b) 7
   (c) 9 (d) 5
3. If an array is declared as double `arr[50];` how many bytes will be allocated to it?
   (a) 50 (b) 100
   (c) 200 (d) 400
4. If an array is declared as `int arr[50]`, how many elements can it hold?
   (a) 49 (b) 50
   (c) 51 (d) 0
5. If an array is declared as `int arr[5][5]`, how many elements can it store?
   (a) 5 (b) 25
   (c) 10 (d) 0
6. Given an integer array `arr[]`; the `i`th element can be accessed by writing
   (a) `*(arr+i)` (b) `*(i+arr)`
   (c) `arr[i]` (d) All of these

True or False

1. An array is used to refer multiple memory locations having the same name.
2. An array name can be used as a pointer.
3. A loop is used to access all the elements of an array.
4. An array stores all its data elements in non-consecutive memory locations.
5. Lower bound is the index of the last element in an array.
6. Merged array contains contents of the first array followed by the contents of the second array.
7. It is possible to pass an entire array as a function argument.
8. `arr[i]` is equivalent to writing `*(arr+i)`.
9. Array name is equivalent to the address of its last element.
10. `mat[i][j]` is equivalent to `*(*(mat + i) + j)`.
11. An array contains elements of the same data type.
12. When an array is passed to a function, C passes the value for each element.
13. A two-dimensional array contains data of two different types.
14. The maximum number of dimensions that an array can have is 4.
15. By default, the first subscript of the array is zero.

Fill in the Blanks

1. Each array element is accessed using a _____.
2. The elements of an array are stored in _____ memory locations.
3. An *n*-dimensional array contains _____ subscripts.
4. Name of the array acts as a _____.
5. Declaring an array means specifying the _____, _____, and _____.
6. _____ is the address of the first element in the array.
7. Length of an array is given by the number of _____.
8. A multi-dimensional array, in simple terms, is an _____.
9. An expression that evaluates to an _____ value may be used as an index.
10. `arr[3] = 10;` initializes the _____ element of the array with value 10.